0

TASK: UU03
CDRL: 05156
19 February 1993

Reuse Library Framework Version 4.1 Administrator Manual

Informal Technical Data

STARS-UC-05156/017/00
19 February 1993

TASK: UC03
CDRL: 05156
February 19, 1993

RLF Administrator's Manual For The SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Reuse Library Framework
Version 4.1
SunOS Implementation

STARS-UC-05156/017/00
February 19, 1993

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0000

Prepared for:

Electronic Systems Center
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Paramax Systems Corporation
Electronic Systems-Valley Forge Engineering Center
70 E. Swedesford Rd.
Paoli, PA 19301
under contract to
Paramax Systems Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UC03
CDRL: 05156
February 19, 1993

Data ID: STARS-UC-05156/017/00

TASK: UU03
CDRL: 05156
February 19, 1993

RLF Administrator's Manual
Reuse Library Framew
Version 4.1
SunOS Implementatio

Principal Author(s):

Timothy M. Schreyer *Date*

Approvals:

Task Manager *Richard E. Creps* *Date*

(Signatures on File)

TASK: UU03
CDRL: 05156
February 19, 1993

RLF Administrator's Manual
Reuse Library Framework
Version 4.1
SunOS Implementation

Change Record:

Data ID	*Description of Change*	*Date*	*Approval*
STARS-UC-05156/004/00	Original Issue	November 1992	*on file*
STARS-UC-05156/017/00	Updates for version 4.1	February 1993	*on file*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Informal Technical Report

4. TITLE AND SUBTITLE

RLF Administrator's Manual

5. FUNDING NUMBERS

F19628-88-D-0031

6. AUTHOR(S)

Paramax Corporation

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Paramax Corporation
1210 Sunrise Valley Drive
Reston, VA 22090

8. PERFORMING ORGANIZATION REPORT NUMBER

STARS-UC-05156/017/00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Air Force
Headquarter, Electronic Systems
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

05156

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This manual is intended for the administrator of a reuse library hosted on the Reuse Library Framework (RLF). Some information on installing RLF and its example libraries may be of interest to the reuse library modeler or user. Specific information on installation can be found in either the **RLF Source Code Release Installation Guide** or the **RLF Binary Release Installation Guide**.

14. SUBJECT TERMS

15. NUMBER OF PAGES

41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unlcassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope

This manual is intended for the administrator of a reuse library hosted on the Reuse Library Framework (RLF). Some information on installing RLF and its example libraries may be of interest to the reuse library modeler or user. Specific information on installation can be found in either the **RLF Source Code Release Installation Guide** or the **RLF Binary Release Installation Guide**.

The reuse library administrator is the person responsible for setting up the reuse library environment so others can use the library to find reusable software assets. This role includes obtaining and installing RLF, testing and configuring the library environment, and possibly handling library user feedback and updating the installation. Updating the installation can include installing new software distributions, adding new assets to existing libraries, changing a library so it is more useful, or setting up a whole new library.

This manual assumes the administrator has a basic understanding of the UNIX operating system and the X Window System. If the administrator plans to install RLF by compiling the actual source of the system, this manual assumes a good understanding of the SunAda or Verdix Ada compilation system. This version of RLF can support execution using Emeraude PCTE v12.3 as an underlying object management system. If RLF is run with PCTE, it is assumed that the user understands PCTE and the Emeraude product, including the ability to construct *esh* scripts.

1.2 Identification

This **RLF Administrator's Manual** provides the information necessary for an RLF reuse library administrator to install, modify, and maintain a reuse library hosted on RLF. More detailed information on the creation and modification of a reuse library's underlying library data model can be found in the separate **RLF Modeler's Manual**. If RLF has been constructed to run with PCTE, then certain guidelines must be followed when modeling for PCTE. PCTE-specific information has been gathered in appendix C.

1.3 Overview

The Reuse Library Framework (RLF) is a knowledge-based system for reuse library construction and operation. By structuring a set of reusable software assets in a knowledge network and representing their descriptions and interrelationships in the network, the RLF increases the library user's chances of finding and extracting the reusable asset that is desired. The knowledge-based representation also helps enhance the user's understanding of the system from which the reusable assets are taken, and allows an "intelligent" help mechanism to aid the user with retrieval of assets.

The remainder of this document is organized as follows. Section 2 explains how to install RLF and test the installation. It also gives some guidelines for setting up the environment in which reuse library users will operate. Section 3 provides some material on the fundamental concepts of RLF including the rationale for the RLF approach to constructing reuse libraries and the concepts involved in a representation of an RLF reuse library. Section 4 discusses library maintenance issues like creating and removing libraries and modifying a library's representation to add or remove new reusable assets or change library operations. Section 5 addresses library administrator issues concerning making the library easier to use and managing the day-to-day operations of the library. Section 6 explains how to get and install new or updated versions of the RLF and how to report errors and get help. This document has three appendices which present an overview of the `Library_Manager` RLF application, the syntax of the `.rlfrc` start-up file and an example, and a discussion of PCTE issues when using RLF and PCTE.

1.4 Notation Used in This Manual

Several different typefaces are used in this manual to notate objects of different kinds. The names of manuals are printed in a **bold** typeface. The names of UNIX tools or utilities are printed in *italics*. The names of directories and files, the text of UNIX shell scripts, environment variable names, and the names of RLF applications are printed in `typewriter` typeface. Examples of Library Model Description Languages (LMDL) and library model categories or objects also appear in `typewriter` typeface.

2 Configuring RLF

2.1 Installing RLF

Included with RLF 4.0 is the **RLF Source Code Release Installation Guide** or the **RLF Binary Release Installation Guide.** It provides all the information necessary to install the RLF. The installation of RLF has been automated through the use of UNIX *csh* scripts. The Installation Guide directs the use of these scripts to set up the RLF.

A successful installation of RLF should provide four applications and a directory containing the representation of the RLF's example libraries. The applications are the `Graphical_Browser` with which the user views reuse libraries and extracts reusable assets, the `Library_Manager` which is used by the library administrator to manipulate libraries, and the LMDL and RBDL translators, `Lmdl` and `Rbdl`, which are used by the library modelers and administrator to create and modify a reuse library's knowledge-based representation. Another tool, `Sndl_to_Lmdl` is delivered with RLF. It translates library model specifications written in Semantic Network Definition Language (SNDL) (the library model domain encoding language for RLF prior to version 4.0) into LMDL. The `Graphical_Browser` application is covered in detail in the **RLF User's Manual,** and the `Library_Manager`, `Lmdl`, `Rbdl`, and `Sndl_to_Lmdl` are described in this manual and the **RLF Modeler's Manual.**

2.2 Testing the Installation

If the installation described in either **RLF Installation Guide** is followed and completes without errors, then the LMDL and RBDL translator applications of RLF have already been tested. To test the `Library_Manager` and `Graphical_Browser` applications, they should be executed and each of the example libraries selected. Exercise the menu hierarchies of each tool and see if the menu actions perform as described in the RLF manuals. There is a sample execution of the `Graphical_Browser` in the **RLF User's Manual**. Check especially that the **Graphical_Browser** can invoke actions and use the advice facility of RLF.

If your installation appears to be incorrect or incomplete, refer to sections 6.3 and 6.4 on reporting trouble and getting help.

2.3 Establishing an Execution Environment

Execution of the RLF applications can be simplified by setting up the environment in which the applications run. One way to set up the environment is to set UNIX environment variables using the UNIX *setenv* command. There are five UNIX environment variables of interest to RLF: **RLF_LIBRARIES**, **RLF_WORKING_DIR**, **RLF_EDITOR**, **RLF_PAGER**, and **XAPPLRESDIR**.

RLF_LIBRARIES can contain the pathname of the directory containing the representation of the RLF libraries. It can be overridden with a command line option. The default place to look for the RLF libraries is in a local directory named **Instances/**.

RLF_WORKING_DIR can contain the pathname of the directory where RLF should conduct file copies and other operating system functions. It is used primarily as the directory to copy assets in the reuse library to when they are extracted from the RLF reuse library. The default directory for this variable is the directory where the RLF tool is invoked.

The **RLF_EDITOR** and **RLF_PAGER** environment variables are the pathnames of the executables the user wishes to use when viewing or editing text files from RLF applications. This allows the use of the editor or pager/viewer with which the user is most familiar. The editor and pager will default to *vi* and *less*, respectively.

XAPPLRESDIR can contain the pathname of the directory containing the file **Browser**, which specifies the X Window System resources for the **Graphical_Browser**. This directory should also contain the directory **bitmaps/**, which holds all the X Window System bitmaps used by the **Graphical_Browser**. By default, the file **Browser** and the directory **bitmaps/** can be found in the **bin/** directory of the RLF installation. So, if the installation is used as built, the full pathname of the **bin/** directory is a good value for the XAPPLRESDIR environment variable and can be set with the UNIX command:

setenv XAPPLRESDIR *RLF_installation_pathname*/**bin**

Table 1 lists the UNIX environment variables of interest to RLF and the applications they affect.

It may be useful for the library administrator to write a script which encapsulates setting some of these variables with execution of the RLF applications. For example, the *csh* script in figure 1 could be used to ensure that library users access the correct reuse libraries and that the **Graphical_Browser** always appears the same on the screen. The script assumes that RLF has

been installed in directory **/libraries/rlf/** and that the reuse library has been constructed in **/libraries/instances/**.

Another way to configure the execution environment for RLF is through the use of an RLF start-up file. Whenever an RLF application begins execution it will attempt to read a file named .rlfrc

Environment Variable	Affected Applications
RLF_LIBRARIES	Library_Manager, Graphical_Browser, Lmdl, Rbdl
RLF_WORKING_DIR	Library_Manager, Graphical_Browser
RLF_PAGER	Library_Manager, Graphical_Browser
RLF_EDITOR	Library_Manager, Graphical_Browser
XAPPLRESDIR	Graphical_Browser

Table 1: RLF Environment Variables

```
#! /bin/csh -f

setenv RLF_LIBRARIES /libraries/instances
setenv XAPPLRESDIR /libraries/rlf/bin
setenv RLF_WORKING_DIR /home/johndoe
Graphical_Browser
```

Figure 1: Example start-up script for the `Graphical_Browser`

and set global variables based on the contents of the file. RLF looks in the local directory first, and then in the directory referenced by the environment variable `HOME`. If RLF does not find a file named `.rlfrc` in either of these places, it skips reading the file.

By providing RLF reuse library users with pre-set `.rlfrc` files configured by the library administrator and only writable by an administrator, the applications can be set to run in a particular fashion for each user. For a summary of `.rlfrc` file syntax and an example `.rlfrc` file, see appendix B.

2.4 Restricting Access

Establishing any resource which will be used and possibly updated by multiple individuals always raises issues of access rights. The RLF's reuse libraries and applications are no exception. When an RLF installation is configured, it should be decided who will be able to run each of the applications and to what extent any library user will be able to retrieve or act on the assets in the reusability libraries. This section addresses some of these concerns for RLF.

2.4.1 Restricting Access to Applications

Access to RLF applications is managed through the operating system where RLF has been installed. Access is granted or removed by the placement of the applications' executable files on the file system and the permissions set on the executable files. For example, the `Graphical_Browser` is the application used by the library users to browse the reuse libraries and extract desired reusable assets. It therefore should be placed on the file system somewhere that it can be accessed by all the expected library users and should be executable by these users. The `Library_Manager` and the LMDL and RBDL translators, on the other hand, since they have the capability to modify the reuse libraries, should not be visible or executable by the average user of the library. Their access should be restricted to the library administrator and any library modelers who will be developing new libraries or modifying existing ones. Although all the RLF applications begin in the same directory after installation, it may be desirable to move them to special areas designed to support only library users or library administrators.

2.4.2 Restricting Access to Actions

The `Graphical_Browser` allows reuse library users to perform actions from menus. These actions typically allow the user to do such things as view the source code of a reusable asset, extract an asset from the library, or preview a design document. These actions are also available to the library administrator from the `Library_Manager`. In addition, other actions, called "privileged" actions, can be executed by the administrator from the `Library_Manager`. These actions might include the ability to edit a design document or source code file, or instruct the library to collect statistical information.

Actions are available in RLF applications because they have been "modeled" as part of the reuse library structure. (More information on modeling actions can be found in section 4.3.2 and the **RLF Modeler's Manual.**) If the specification of an action in LMDL includes the keyword "privileged" then this action will be available only from the `Library_Manager` and not the `Graphical_Browser`. In this way, access to actions which are designed for the library administrator or which may alter the reuse library can be made available to the library administrator or modeler but prohibited from the average reuse library user.

Also, since actions are sometimes invoked by having the operating system execute a command string, specifying a tool in the command string which is not available to the user or which is not executable by the user would also restrict that action from the user. This method of restricting actions to the user is not suggested, however, since from the user's standpoint it will appear that the `Graphical_Browser` is not operating correctly when a message box appears reporting that the action was not invoked successfully.

2.4.3 Restricting Access to Assets

The library administrator may find it necessary to restrict access to some or all of the assets in a reuse library. Although one of the primary purposes of the reuse library is to allow users to extract reusable assets for reuse, there may be assets in the library which the administrator does not want the user to extract, or assets that only some users should be able to extract. Also, there may be some sort of formal check-out procedure for configuration management or reuse library management which must be conducted before a library user can obtain an asset. Another issue is viewing and extraction of assets which may be classified or company proprietary.

Much of this restriction can be done by tailoring the default extract action modeled in the RLF example networks. (More information on modeling actions can be found in section 4.3.2 and the **RLF Modeler's Manual.**) The default extract action does a simple UNIX copy using *cp* from the reuse library to the user's working directory. This action could be modified to call a script which, for example, could do a configuration management check for existing locks, a "check-out" for the user if a lock doesn't exist, and then a copy to the user's directory. This script would prevent the extraction of a component if someone else had already extracted it and a lock was remaining. In another example, the extract action could be modified to send a mail message to the library administrator containing the name of the desired asset, and then the administrator could mail a copy of the asset back to the user if the user was allowed to extract that asset. The ability of the action invocation mechanism to execute any script leaves the possibilities for complex extract actions very open.

Similarly, restricting the ability to view an asset can be implemented by modifying or replacing the default view action delivered with the RLF example libraries.

The permissions of the file system and the location of assets may also affect the access to library assets, either by design or by accident. If the view or extract action of the library requires a tool which the user cannot invoke or tries to read an asset for which the user does not have permissions, the action will fail presenting the user with a message box reporting the failure. It is important to make sure files referenced by actions in the library are available to the user for viewing and extraction where desired. Likewise, making files available on the file system to only certain users can be used as a kind of access control.

3 RLF Fundamentals

3.1 Domain Model Approach

RLF's approach to managing a reuse library is based on the principle that a highly-structured reuse library will be easier to browse and understand. The structure of an RLF library is provided by a knowledge network which not only classifies the assets in the library in a hierarchy from general to most specific, but also describes the relationships between assets and the part they may play in the composition of a larger system.

When an RLF library model which describes this knowledge network is constructed, the first step is to identify the area common to all the assets to be available in the reuse library. This area is called a "domain." The process of defining the domain is called "domain analysis," and the process of encoding that domain into some sort of structure is called "domain modeling." These activities in general can become very complex and it is beyond the scope of this manual to fully describe them here. Information on modeling a reuse library for RLF is given in the **RLF Modeler's Manual**.

The "domain model" is the final product of domain modeling. By capturing the domain model of the reusable assets in the library as an RLF knowledge network, the level of understanding of the assets in the library increases significantly. This in turn improves the chances that an asset extracted from the RLF library will be immediately useful to the library user. The key to effective reuse is to minimize the time taken to find and retrieve the asset to be reused, and to increase the chances that the asset can be reused without much alteration. The domain model approach ensures that there is enough information in the library structure to meet these goals.

3.2 RLF Concepts

The following five subsections describe the entities that compose an RLF library model. These descriptions are provided so that the library administrator will understand enough of the library

```
root category Thing is
end root category;

category Algorithms ( Thing ) is
end category;

category "Search Algorithms" ( Algorithms ) is
end category;
```

Figure 2: Some Examples of Categories

model's composition in order to be able to interpret and modify it. Complete semantic d[illegible]ions of the fundamental RLF entities can be found in the **RLF Modeler's Manual**. Example of entities are given in fragments of the Library Model Definition Language (LMDL). A complete description of LMDL syntax and semantics can also be found in the **RLF Modeler's Manual**.

3.2.1 Categories

Categories are general descriptions of a kind of thing. They can be thought of as a classification of what a thing is. Examples of categories might be `algorithm`, `file`, or `table search`. Categories can be very general or very specific. In a library model, categories are arranged in a hierarchy with the most general category at the highest level and more specific categories below it, with the most specific categories at the lowest level. This hierarchy is arranged so that every category which appears below a given category in the hierarchy is a more specific description of that category. For example, `search algorithm` would appear below `algorithm` but above `table search`. The most general category in the library model is called the "root category." Examples of the LMDL definitions of some categories in the domain of search and sort algorithms appear in figure 2.

3.2.2 Objects

RLF objects represent actual things instead of classifications of things which categories represent. Examples of objects might be a particular `quick sort` or `binary search` algorithm which is an asset in the reuse library. Objects are always associated with the most specific category which describes them and can be thought of as appearing below these categories in the hierarchy described in section 3.2.1. Objects' attributes are the goal of reuse library users looking for reusable assets. The whole library is established to aid the user in the location of objects so that the object's valuable attributes can be extracted from the library and reused. Attributes are described in section 3.2.4. Example LMDL definitions of some objects in the sort and search domain are given in figure 3.

3.2.3 Relationships

RLF relationship entities are used to describe categories and objects and express the associations between different categories or objects. Relationships are defined at the category they describe and

```
object Ada ( "Source Language" ) is
end object;

object "Example Quicksort" ( Quicksort ) is
end object;

object "Example Binary Search" ( "Binary Search" ) is
end object;
```

Figure 3: Some Examples of Objects

are also valid for all categories or objects below that category in the hierarchy. Relationships express the idea that categories have other categories which describe them. For example, an **algorithm** category might have relationships which describe what language the algorithm is written in, what type of data structures it operates on, or what the worst case performance of the algorithm is. Relationships can also express that a category or object is composed of other categories or objects. For example, a **book** category might have a relationship which shows that a book is composed of chapters, with the **chapter** category defined elsewhere in the library model.

Relationships have a name which helps describe the relationship, an "owner" which is where the relationship is defined, a "type" which is the category describing the owner, and a range of values called the "cardinality" of the relationship. For example, in a relationship which shows that a book is composed of one to any number of chapters, the name might be **has_chapters**, the owner **book**, the type **chapter**, and the cardinality one to infinity. When a relationship exists between two objects, it is said to be "filled," and the object that is the type of the relationship is said to "satisfy" the relationship.

Relationships can be narrowed or "restricted" at subcategories or objects of the owner category. The type category can be made more specific or the cardinality made smaller. Relationships, like categories, can become more specific lower in the hierarchy, but never more general. When a relationship has been restricted, its new type and cardinality are the ones valid for all categories or objects in the hierarchy below the one where the relationship is restricted. Restriction and more complex operations on relationships are discussed in the **RLF Modeler's Manual.**

LMDL examples of some relationships and restricted relationships in the sort and search algorithms domain are given in figure 4.

3.2.4 Attributes

RLF attributes tie an abstract library model to the actual data and reusable assets in the library. Attributes can be integers, strings of characters, or files. They are associated with categories or objects in the library model. The attributes are given names so they can be referenced by RLF actions (described in section 3.2.5) and be viewed, extracted, or otherwise manipulated. Attributes are only valid at the category or object where they are defined and are not available from subcategories or objects like relationships. Some examples of attributes defined in LMDL from the sort

```
category Algorithms ( Thing ) is
   relationships
      is_written_in (0 .. infinity) of "Source Language";
      works_on (0 .. infinity) of "Data Structure";
      has_best_case_of (0 .. 1) of Performance;
      has_avg_case_of (0 .. 1) of Performance;
      has_worst_case_of (0 .. 1) of Performance;
      has_size_of (0 .. 1) of "Lines of Code";
   end relationships;
end category;

category "quick sort" ( exchange_sorts ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
end category;

object example_quicksort ( "quick sort" ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Quadratic;
      has_size_of (0 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N^2" satisfies has_worst_case_of;
      "Twenty-Four" satisfies has_size_of;
   end fillers;
end object;
```

Figure 4: Some Examples of Relationships

```
category Extract ( Action ) is
   attributes
      string is "Extract Asset";
   end attributes;
end category;

category Quicksort ( "Exchange Sorts" ) is
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;

object "Example Quicksort" ( Quicksort ) is
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
      file source is "sort_and_search/quick_sort_.a";
      string size_of is "24";
   end attributes;
end object;
```

Figure 5: Some Examples of Attributes

and search algorithms domain appear in figure 5.

3.2.5 Actions

RLF actions let the library user manipulate attributes or do other things like mail messages to the library administrator, collect library information, or get LMDL representations of part of the library model. Actions are the chief way that users will obtain copies of the reusable assets in the library. Actions must be specified by the library modeler or administrator in the LMDL specification of the library model in order for the user to be able to invoke them. A set of common, useful actions is already modeled into each of the example library models and can be reused in any new reuse library LMDL specifications.

Actions are associated with categories or objects in the library model. These are available to subcategories and objects below the category or object where they are defined like relationships are. An action has a name, an "action category" which is a category elsewhere in the library model which describes the action, a list of "action targets" which are the names of attributes at that category or object upon which the action will act, and a list of "action agents" which are the names of attributes at that category or object which will modify the action when it is invoked. Actions can also be privileged. Privileged actions can only be invoked from the `Library_Manager` application and not the `Graphical_Browser`. They are intended as actions which a library administrator would like to invoke that shouldn't necessarily be available to the average user.

Actions can be narrowed at subcategories or objects below the category or object where they are

```
category "Insertion Sorts" ( "Internal Sorts" ) is
   attributes
      file desc_source is "sort_and_search/insertion_sort_desc";
   end attributes;
   actions
      "Read Description" is "Display Description" on desc_source;
   end actions;
end category;

object "Example Quicksort" ( Quicksort ) is
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
      file source is "sort_and_search/quick_sort_.a";
      string size_of is "24";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
      "Extract Source" is Extract;
   end actions;
end object;
```

Figure 6: Some Examples of Actions

defined, in a manner similar to relationships. A non-privileged action can be made privileged or an action category more specific. Once an action has been made privileged, it cannot be made non-privileged lower in the hierarchy.

Action categories are defined in the library model just like any other category. Action categories appear in the hierarchy below a reserved category named `Action`. A string attribute at an action category is used to invoke the action. RLF currently supports two types of actions, `"System String"` and `"Ada Procedure"`. A `"System String"` type action uses the action category's string attribute as a string to be executed in the operating system shell. An `"Ada Procedure"` type action uses the action category's string attribute to match a built-in Ada procedure to call when the action is invoked. A more detailed description of action category definitions can be found in section 4.3.2. More information on action semantics and action invocation can be found in the **RLF Modeler's Manual.**

Some example actions from the sort and search algorithms library model are shown in LMDL form in figure 6. Some action category definitions are shown in figure 7.

3.3 Library Advice

RLF has a knowledge-based component intended to aid the user in selecting reusable assets if the user is not an expert in the domain of the library or is having trouble choosing between assets.

```
category "Action Definition" ( Thing ) is
end category;

   category Action ( "Action Definition" ) is
      relationships
         has_action_type (1 .. 1) of "Action Type";
      end relationships;
   end category;

      category View ( Action ) is
         restricted relationships
            has_action_type of "System String";
         end restricted;
         attributes
            string is "xterm -e $RLF_PAGER ## &";
         end attributes;
      end category;
```

Figure 7: Example Action Category Definitions

This aid is called library advice. Advice can be available from any category or object in the model. Advice is modeled in the Rule-Base Definition Language (RBDL) and each advice module, called an "inferencer," is then attached to a category or object in the LMDL specification of the library model. If the library administrator or modeler does not model and attach the advice modules to the library, there will not be any advice available to the library user.

Advice typically will ask the user questions about the asset desired, and then will make deductions about which areas of the library model are most appropriate to search and optionally relocate the user there automatically. When advice is selected from a menu in an RLF application, a window will pop up. The question and answer session of the advice module is conducted in this window, adjusting the current node in the main application as it executes.

Modeling and providing good advice for a library will greatly enhance its usefulness. It can make a complex library model understandable to a novice in the domain of the library who might not be able to use the library at all without advice. Modeling good advice can be difficult, however, and more information on this is presented in the **RLF Modeler's Manual**. Example RBDL is given there. Figure 8 in this manual shows how an inferencer is attached to the library model in the library model LMDL specification.

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;

attach inferencer quicksort to Quicksort;
```

Figure 8: An Example of Connecting Library Advice to the Library Model

4 Library Maintenance

This section outlines for the library administrator the steps to follow to make changes to the library model that structures the reuse library. The outlines make reference to running the LMDL translator and the examples appear in LMDL. This section does not attempt to describe LMDL in full or cover all the functionality of the LMDL translator. These areas are described in the **RLF Modeler's Manual.** Examples in this section assume that the RLF_LIBRARIES environment variable (further described in section 2.3) has been set to the directory containing the reuse library being modified and that the LMDL translator, Lmdl, appears in the library administrator's path.

4.1 Build New Libraries

New libraries are constructed by running the LMDL translator on the LMDL specification of the library model. Assuming that spec.lmdl is the LMDL specification of the library model, the command

```
Lmdl spec.lmdl
```

issued at the UNIX shell prompt will construct the new library in the directory to which the environment variable RLF_LIBRARIES is set. If a library with that name already exists, it will be overwritten.

4.2 Remove Libraries

Libraries are removed using the Library_Manager application. See appendix A.

4.3 Modify Existing Libraries

This subsection describes the various library modifications which a library administrator is likely to make. Most modifications require the editing of the library's LMDL library model specification. After this editing has occurred the specification must be retranslated with the LMDL translator, **Lmdl**. Assuming that the library's LMDL library model specification is in a file named `spec.lmdl`, and that the **RLF_LIBRARIES** environment variable specifies the directory where the library has been built, then running the LMDL translator is accomplished by issuing the following command at the UNIX shell prompt:

```
Lmdl spec.lmdl
```

When this command completes successfully, the library model specification has been "retranslated" and any changes made while editing the specification will now be realized when the library is viewed with an RLF application.

Note: If the only changes made to the library's library model specification were changes to the integer, character string, or file attributes of categories or objects, the LMDL translator should be invoked with the **-state** command line option when retranslating. This significantly reduces the amount of time required to retranslate the library model specification.

4.3.1 Assets

Assets are represented in the library model by integer, character string, and file attributes of categories and objects. In most cases, assets will be file attributes of objects. This subsection describes operations on the library model affecting assets. When using the PCTE version of RLF there are certain restrictions to the name and location of file attributes in the library model. See appendix C for specific information on these restrictions.

Add New Assets

A new asset is added by defining an attribute at the object which best represents the asset. Sometimes an object must be created to represent the asset if one does not currently exist. After these changes, the LMDL specification of the library model must be retranslated for the new attributes and objects to become part of the library.

Suppose there is a reusable quick sort implementation which is to be added to the sort and search algorithms library. By browsing the library with the **Graphical_Browser** or the library model specification with an editor, the quick sort algorithm category is located and determined to be the most representative of the new asset. The quick sort category appears as such:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
```

```
      attributes
         file desc_source is "sort_and_search/exchange_sort_desc";
      end attributes;
   end category;
```

First an object is created by editing the specification to include the following:

```
   object "Example Quicksort" ( Quicksort ) is
   end object;
```

Next the actual asset is attached to the object by defining a file attribute of the object. The file containing the asset is given a pathname relative to the directory Text, which is a first-level subdirectory below the directory where the library representations exist. The object definition now looks like this:

```
   object "Example Quicksort" ( Quicksort ) is
      attributes
         file source is "sort_and_search/quick_sort_.a";
      end attributes;
   end object;
```

This is the minimal definition which will attach the asset to the library. The asset's file must be copied into the appropriate directory in the library directory structure. For our example, the asset would be copied to **$RLF_LIBRARIES/Text/sort_and_search/quick_sort_.a**. Then when the LMDL specification had been retranslated, the asset would be visible from the reuse library.

When adding an asset, however, the library modeler and administrator should also describe the object representing the asset as fully as possible. This includes restricting and filling any relationships that may have been defined anywhere in the hierarchy directly above the object on a direct path to the root category. Also, any actions valid at the object or desired just at the object need to be defined, and any additional attributes which the objects has must also be defined. The final definition of the new asset's object might look like this:

```
   object "Example Quicksort" ( Quicksort ) is
      restricted relationships
         is_written_in (1 .. 1) of "Source Language";
         works_on (1 .. 1) of "Data Structure";
         has_worst_case_of (1 .. 1) of Quadratic;
         has_size_of (0 .. 1) of Number;
      end restricted;
      fillers
         Ada satisfies is_written_in;
         Array satisfies works_on;
         "N^2" satisfies has_worst_case_of;
         "Twenty-Four" satisfies has_size_of;
      end fillers;
      attributes
         file desc_source is "sort_and_search/exchange_sort_desc";
         file source is "sort_and_search/quick_sort_.a";
         string size_of is "24";
      end attributes;
      actions
         "View Code Size" is "Display Integer" on size_of;
         "View Source" is View on source;
         "Extract Source" is Extract;
      end actions;
   end object;
```

Remove Assets

Assets can be partially or completely removed from the library by editing the library model specification and retranslating it. To partially remove an asset, simply remove the file attribute at the object which represents it. After retranslation, the object describing the asset can still be examined, but the actual asset cannot be extracted since it is no longer an attribute of the object. To completely remove an asset from a library, remove the entire object definition representing the asset from the library model specification and then retranslate. After this there will be no reference to the asset in the library.

Assets can also be removed by deleting the attributes associated with them from within the **Library_Manager** application. See appendix A.

Swap in New Versions of Assets

New versions of assets can be made available to the library without having to having to go through the add asset procedure in section 4.3.1. This should only be done if the old version of the asset should no longer be available in the reuse library. For more information on version control see section 5.1.

The simplest way to swap in a new version of an asset is to write its file over the one already defined for the asset. This is usually a file attribute of the object representing the asset. For example, if it is desired to swap in a new version of a quick sort asset defined by

```
   object "Example Quicksort" ( Quicksort ) is
      attributes
         file source is "sort_and_search/quick_sort_.a";
      end attributes;
   end object;
```

the new file could be copied on top of **quick_sort_.a** in the library and be immediately available to the reuse library user.

A different way to do this would be to change the filename specified in the attribute of the object and copy the asset to that file. For instance, the object definition above could be changed to

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file source is "sort_and_search/new_quick_sort_.a";
   end attributes;
end object;
```

and then the asset's file copied to new_quick_sort_.a. It is important to remember that the pathnames of file attributes are relative to the directory Text/ which is a first-level subdirectory below the one specified in the RLF_LIBRARIES environment variable. This approach to swapping in a new version of an asset requires that the library model specification be retranslated with the LMDL translator, Lmdl, before the new version of the asset will be available to the library user.

Restrict Access to Assets

Information on restricting access to assets can be found in section 2.4.3.

4.3.2 Actions

The RLF action mechanism, in company with library model attributes, bridges the gap between the library MODEL of the library and the actuals ASSETS that are in the library. Various actions in the library model allow the user to view assets in different ways and then extract them. This sections describes how library model actions are modified by the library administrator and modeler in order to change the behavior of an RLF resue library. When using the PCTE version of RLF, other modeling restrictions and conventions come into play. See appendix C for details on RLF and PCTE.

Add New Actions

New actions are added by modifying the library model in two areas. One section of each RLF library model contains a sub-model rooted at the reserved category "Action Definition". "Action Definition" has two subcategories named Action and "Action Type". The sub-model rooted at Action contains descriptions of all the actions that can be available at other categories and objects in the library. The actions described in this sub-model are called "action categories." Although an action category can have other relationships and attributes which describe it, the most important parts of the action category are its has_action_type relationship which it inherits from Action and restricts locally and a string attribute which is used to invoke the action.

Below the "Action Type" category are sub-categories which describe the different types of actions available within the RLF reuse library. There are currently two types of actions supported, "System String" and "Ada Procedure". A "System String" type action uses the action category's string attribute as a string to be executed in the operating system shell. An "Ada Procedure" type action uses the action category's string attribute to match a built-in Ada procedure to call when the action is invoked. It is expected that additional types of RLF actions will be added in the future by adding additional subcategories to "Action Type". Possibilities include a message passing action and actions tailored to the environment in which the reuse library operates. New action types will interpret the action category's string attribute in the appropriate way for that type of action.

The category Action defines a relationship named has_action_type with a type of "Action Type". At each action category below Action this relationship is inherited and should be restricted to a more specific "Action Type". The following excerpt from the sort and search algorithms library LMDL specification shows a part of the "Action Definition" sub-model including Action, "Action Type", and some of their subcategories.

```
category "Action Definition" ( Thing ) is
end category;

   category "Action Type" ( "Action Definition" ) is
   end category;

      category "System String" ( "Action Type" ) is
      end category;

      category "Ada Procedure" ("Action Type") is
      end category;

   category Action ( "Action Definition" ) is
      relationships
         has_action_type (1 .. 1) of "Action Type";
      end relationships;
   end category;

      category View ( Action ) is
         restricted relationships
            has_action_type of "System String";
         end restricted;
         attributes
            string is "xterm -e $RLF_PAGER ## &";
         end attributes;
      end category;
```

This example also shows the definition of a View action for the library. View is a "System String" type action which will have the string attribute "xterm -e $RLF_PAGER ## &" executed in an operating system shell when it is invoked. The restriction of has_action_type's type to "System String" is required so that RLF will know how to invoke the action correctly.

This example action definition also introduces substitution markers. When an action is of type "System String", the string attribute at the action category can contain special series of symbols which can be used to parameterize the action when it is invoked. A special marker, "##", in an action category's string attribute holds the place in the string where an argument to the action, called the "action target," will be substituted. The action target is supplied at the category where the action is available to be invoked. An action category's string attribute can also contain markers for "action agents" which are also supplied at the category or object where the action is available. Action agent substitution markers have the form "%%n", where n is a numeral from 1 to 9. For more information on action targets and agents, and how system string actions are invoked, consult the **RLF Modeler's Manual**.

RLF supports four built-in Ada procedure actions: `Import`, `Export`, `Extract`, and `''Display Attributes''`. These actions are modeled by action categories in the action sub-model which have restricted the **has_action_type** relationship to type `"Ada Procedure"`. These action categories can be referenced from action definitions in the main part of the reuse library domain model. Any new actions of the `"Ada Procedure"` type which are `Import` or `Export` actions should probably be defined as privileged actions, since these operations are primarily library administrator operations and `Import` types actions may modify the reuse library model.

The first step to adding a new action to an RLF reuse library is either to locate the desired action in the action sub-model section of the library's LMDL specification or to create the appropriate action category in the action sub-model. The same care should be taken in modeling action categories that is taken modeling parts of the main library model. Action categories which are subcategories of other action categories should be more specific forms of those categories. If the action being added is not related to any of the pre-existing example actions modeled with action categories, then a new action category describing the action should be defined as a direct subcategory of the category `Action`.

The new action category definition should restrict the **has_action_type** relationship inherited from `Action` to have type `"System String"` or `"Ada Procedure"`. If the action type is `"System String"`, then a string attribute should be defined at the action category which is the string to be executed in an operating system shell when the action is invoked. Substitution markers should be used in the string where action targets or agents will appear when the action is invoked. Action target, agents, and invocation is discussed in detail in the **RLF Modeler's Manual**. If the action type is `"Ada Procedure"`, then the string attribute at the new action category should be one of `"Import Asset"`, `"Export Asset"`, or `"Extract Asset"`, which are the built-in Ada procedure actions available.

An example of a new action category which will print a file associated with a category or object in the library model follows:

```
category Print ( Action ) is
-- this action category describes a general print action
   restricted relationships
      has_action_type (1 .. 1) of "System String";
   end restricted;
   attributes
      -- ## marks the file to be printed
      -- %%1 marks the UNIX print command to use
      -- %%2 marks any options to the print command
      -- also run the action in the UNIX background
      -- so the RLF application continues
      string print_command is "%%1 %%2 ## &";
   end attributes;
end category;
```

This LMDL fragment defines an action category named `Print` which describes a `"System String"` type action which prints its action target file using two action agents for the print command and any print command options. When an action which has `Print` as its action category is invoked, it will gather the required action target and agents from the category or object where it is invoked, process the action category's string attribute replacing the substitution markers with their actual values, and then executing the final string in an operating system shell.

The other area of the library model which is modified to add new actions is the category definitions in the main library model. Actions are defined within a category very much like relationships, and are similarly available at subcategories and objects of the category or object where they are first defined. Once the action category is located or created in the action category sub-model, it can then be referenced at the categories where it will be available. Suppose the library administrator wants the library user to be able to print the source code of a quick sort implementation which is a reusable asset in the library. The print action could be defined at the object representing the quick sort implementation like this:

```
object "Example Quicksort" ( Quicksort ) is
   actions
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

This defines an action named **"Print Source"** at the object and tells RLF that the action is described by the action category named **Print** and will operate on the local file attribute named **source**. **source** is the action target. The action will also use the action agents **print_command** and **print_options** to modify the action invocation. If these attributes are defined like this:

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file source is "sort_and_search/quick_sort_.a";
      string print_command is "lpr";
      string print_options is "-Pprinter1";
   end attributes;
   actions
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

then, assuming the definition of the **Print** action category given above, when the action is invoked at the **"Example Quicksort"** object in the library, the file **sort_and_search/quick_sort_.a** will be printed using the *lpr* command on the printer specified in the option **"-Pprinter1"**. (Note: The file name is relative to the **Text/** subdirectory, which is a first-level subdirectory below the directory specified in the **RLF_LIBRARIES** environment variable.)

When defining new actions at categories in the main library model, it is useful to remember that actions can process a list of targets. For instance, suppose the library administrator wished to provide an action which would print the abstract, performance study, and source code for a particular quick sort implementation in the sort and search algorithms library. It would be best to use a list of targets so one action invocation by the user would print all the associated files. One solution in LMDL could look like this:

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file abstract is "sort_and_search/quick_sort_abstract";
      file performance_study is "sort_and_search/quick_sort_perf";
      file source is "sort_and_search/quick_sort_.a";
      string print_command is "lpr";
      string print_options is "-Pprinter1";
   end attributes;
   actions
      "Print All Data" is Print on abstract, performance_study, source
         with print_command, print_options;
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

This would provide a `"Print Source"` action to just print the implementation's source and a `"Print All Data"` action which would print the implementation's abstract, performance study, and source. When the `"Print All Data"` action is invoked, it will iterate over the list of targets performing the action described by action category `Print` for each file in the list. Again, more details on action invocation semantics and modeling appears in the **RLF Modeler's Manual**.

When new action categories have been added and new actions defined in the main library model which reference them, the library model definition must be retranslated by the library administrator using the LMDL translator, `Lmdl`, in order for the actions to be available from the RLF applications.

Modify Actions

RLF `"System String"` type actions are modified by altering the action command string which is defined in the library model at the action's action category. The action category is a category in the action sub-model which is rooted at the reserved category `Action`. It describes the action and provides the action command string which is executed when the action is invoked. More information on the action sub-model, action categories, and action types can be found in the previous section on adding new actions.

The `View` action for the sort and search algorithms library is described at its action category as follows:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ## &";
   end attributes;
end category;
```

If the library administrator wanted to modify the view action so that it no longer ran in the UNIX background and halted the RLF application instead, then the library model definition could be modified like so:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ##";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, **Lmdl**, the view action, when invoked, would execute its new behavior and halt the RLF application until the view was complete

Similarly, if the library administrator wished to change the **View** action so that it used a specific editor instead of using the **RLF_PAGER** environment variable to view the asset, then the view action category could be changed to this:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e /usr/ucb/view ##";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, **Lmdl**, the view action, when invoked, would execute its new behavior and the asset would be viewed with *view* instead of the pager found in **RLF_PAGER**.

If the library administrator wished to do more complex operations when viewing an asset such as collecting metrics or doing configuration management, then the view action's operations could be put into a UNIX shell script, and the library model of the view action category modified as follows:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "asset_view.csh ## &";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, `Lmdl`, the view action, when invoked, would execute the *csh* shell script named `asset_view.csh` passing the name of the file to the script as a parameter. This script would also execute in the UNIX background allowing the RLF application to continue.

Note: To save time when modifying a library model's action categories, if the only changes made were changes to the string attributes at action categories, the LMDL translator should be invoked with the `-state` command line option when retranslating.

Restrict Access to Actions

Information on restricting access to actions can be found in section 2.4.2.

Remove Actions

Actions can be removed from the library model and thus the library in two ways. To make an action unavailable from certain categories or objects, but still present to others, the action definitions at the categories or objects can be removed. Since the action category for the action still exists in the action sub-model, it will still be available to categories and objects where the action has been kept. To remove the action from the library entirely, the action category description of the action should be removed from the action sub-model in the library model, and then all action definitions which reference that action's action category should be deleted. Both these methods for removing actions will only be evident after the library model specification had been retranslated by the LMDL translator, `Lmdl`.

4.3.3 Advice

RLF library advice is provided through the RLF's inferencing subsystem AdaTAU. Library advice is modeled in the Rule Base Definition Language (RBDL). Complete information on modeling library advice for RLF reuse libraries is provided in the **RLF Modeler's Manual**. This section addresses how the library administrator can manipulate advice attached to an RLF library.

Add New Advice

Once new advice has been modeled and built as described in the `RLF Modeler's Manual`, it can be attached to an RLF reuse library by editing the library's LMDL library model specification and then retranslating it. When making changes to the library model specification to change library advice, the LMDL translator, `Lmdl`, should be invoked with the `-state` command line option. This significantly reduces the amount of time that it takes to retranslate the library model.

Suppose the library modeler has produced a new advice module, or "inferencer," for a category or object in the library model and the library administrator wishes to make the inferencer available to the RLF applications. If the library administrator wishes to add the inferencer to the `Quicksort` category of the sort and search algorithms library, the following line would be added to the LMDL library model specification:

```
    attach inferencer quicksort to Quicksort;
```

Now, if the inferencer has been named "quicksort" in its RBDL specification and has been built using the RBDL translator, Rbdl, then once the modified library model specification has been retranslated the library advice contained in the quicksort inferencer will be available at the category Quicksort from the RLF applications.

Modify Advice

Modifications to the actual library advice at a category or object is in the role of the library modeler and is described in the **RLF Modeler's Manual**. Library advice can be modified transparently by retranslating the RBDL specification that defines the inferencer containing the advice. No changes are necessary to the library's library model specification unless the name of the inferencer has changed. If the name has changed, the library model specification should be edited to reflect the name change and then retranslated to have the change installed. Running the LMDL translator, Lmdl, with the -state command line option is sufficient in this case.

Remove Advice

Library advice can be removed by removing the inferencer attachment in the library's library model specification and then retranslating the specification with the LMDL translator, Lmdl. For example, if library advice has been attached to the Quicksort category of the sort and search algorithms library with the following LMDL:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;

attach inferencer quicksort to Quicksort;
```

then removing the attachment of the inferencer and leaving the Quicksort category definition like this:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;
```

will remove the ability to access advice at this category once the library model specification has been retranslated. Also, since only a portion of the model defining library advice has been removed, it will be sufficient and quicker to retranslate the specification using the LMDL translator's `-state` command line option.

5 Manage Library Use

The library administrator is responsible for maintaining the performance and day-to-day operations of an RLF reuse library. The administrator is also responsible for improvements to the library whether they come from user feedback or the library modeler or other sources. This section describes some issues of concern to the administrator of a reuse library and suggests ways to address them with RLF. Most RLF suggestions will have to do with enhancing the default actions of the library model.

5.1 Managing Assets

Many of the assets in a library will not be final products and may even be under modification by library users. This means that some assets will have to be replaced with more up-to-date or efficient versions of the asset periodically. Others may have to have their extraction controlled so only one person can have a copy at a time. Under many circumstances, issues of asset management will need to be considered.

Updating versions of assets can be simple or complex. The easiest way to do updates is to copy the new version over the old one, and the library will reference it just like the old version. Or alternately, the file attribute of the object representing the asset could be changed to reference the new version. This would require the library model to be retranslated to make the update. Translation of the specification with the `-attrs` command line argument would be sufficient and quickest. Version updates can be complex if it is desired that old versions of assets remain in the library. New objects would have to be created in the library model specification and any relationships referring to version numbering would have to be filled. Then the specification would have to be retranslated in its entirety.

The library administrator should maintain some sort of configuration management on the library's assets so that old versions can be recovered, and differences between old and new versions can be revealed. The extent of this activity and its visibility to the user and the reuse library is up to the library administrator and may be affected by the nature of the assets in the library. More on asset management can be found in section 2.4.3.

5.2 Collecting Metrics

It is important to know how the reuse library is being used in order to determine how to improve it. This might require information about which assets are extracted most often, how often users are browsing a library and for how long, or which categories are visited most frequently. Many issues on collecting metrics for reuse libraries are open topics, but RLF can be adapted to collect metrics on many kinds of library use. Following are some ideas on how to collect metrics with RLF.

One way is to modify RLF actions to record entries in a log whenever they are invoked. Since RLF actions can invoke full UNIX shell scripts, there are many ways this can be done. Modifying the default extract action to write a log entry before copying the asset to the user could provide metrics on which assets are most popular by processing the log. Similarly, a modified view action could be used to see what assets users are interested in enough to look at. This log could be compared with the extraction log to gather even more information on what assets people looked at but didn't bother to extract for some reason.

Another way to record metrics on RLF library use would be to establish and maintain an information-gathering start-up script (for more on this see section 2.3). This start-up script could collect user, time-of-day, session length, and other information and write it into a log file or mail it to the library administrator. By careful setting of file system permissions, the user may never need to know that session information is being taken, or that the `Graphical_Browser` is running from a script and not directly.

5.3 Coordinating User Feedback

To best serve the needs of the reuse library user, the library administrator should provide some opportunity for user feedback. Knowing what problems users have and what they like and dislike about the reuse library can lead to improvements which can lead to greater library use and usefulness. Collecting user input can be handled by having users fill out comment forms or trouble reports, but user feedback is much more likely if it is automated and quick.

RLF can automate user feedback through its action mechanism. The library modeler or administrator could include an action which would appear at every category or object which would mail the library administrator a message which the user could enter while still running the `Graphical_Browser` and the question, problem, or suggestion is still fresh. Help actions could also be modeled this way. The action could mail the administrator the file created as the result of spawning a text editor from the `Graphical_Browser`.

5.4 Optimizing Performance

A reuse library will not be useful unless it changes to meet the requirements of the users. A library that is too hard to use or too complex will not be used. That is why it is necessary for the library administrator to worry about library performance. Many issues of performance can be addressed in the library model and the installation, outside of any RLF application performance issues.

The design of the library model structuring the reuse library can affect performance. An overly large or overly complex library model may dissuade users from using the library. Although it may seem better to have more information at hand for the user, it is possible to have the library model too detailed. Trimming the library model specification and retranslating may build a library that is easier to use and understand.

Similarly, the look and feel of the `Graphical_Browser` may dissuade people from using the library. The browser is highly configurable through the X Window System resource file named `Browser` and the `bitmaps/` directory, both of which appear in the `bin/` directory when RLF is installed. Tailoring the size and feel of the browser may make it more attractive to users at particular sites. Also, establishing a start-up script as in section 2.3 can allow library administrator control over the look of the application. Smaller models also run more quickly in the `Graphical_Browser` and that may encourage users where a slow browser running a very large detailed model would discourage them.

The best ideas for optimizing the performance of the reuse library comes from monitoring and responding to library use metrics and user feedback. Knowing how the library is being used, or failing to be used, is the most important information about how to optimize it.

6 Software Maintenance

This section describes how your RLF installation is supported and explains how you can get new releases of RLF and install them. The installation of RLF includes three forms:

- A registration form (in file *Registration_Form*) that should be filled out and returned so that the RLF user base can be notified of product upgrades and other important product news.
- A Program Problem Report (in file *Problem_Report*) that is used to identify any specific problems encountered in installing and using the software.
- A New Feature Request (in file *Feature_Request*) that is used to describe specific enhancements that should be incorporated into the product.

There are three mailing lists established to handle RLF discussion, requests, or problems:

- **rlf@stars.rosslyn.paramax.com**
  This list provides a public forum for discussing RLF issues. Members of this list receive all messages sent to the list and may respond accordingly.
- **rlf-request@stars.rosslyn.paramax.com**
  Completed registration forms are sent to this address, as well as requests to be added to or deleted from the rlf list (NOTE: do NOT send add or delete requests to the rlf list itself).
- **rlf-bugs@stars.rosslyn.paramax.com**
  Completed Program Problem Reports and New Feature Requests are sent to this address.

6.1 Getting Software

The RLF is available by anonymous FTP to `stars.rosslyn.paramax.com` in directory `pub/RLF/`. If AFS access is available, RLF can be found in directory `/afs/stars.reston.unisys.com/see/rlf/4.0/` on cell `stars.reston.unisys.com`. (Note: In the future, this AFS address may change due to Paramax Internet reorganization.) It is delivered in binary and source form, so it is not necessary for the RLF source in the `code/` directory to be retrieved. If FTP or AFS are not available, sending electronic mail to the **rlf-request@stars.rosslyn.paramax.com** mailing list will handle your request. If electronic mail is not an alternative, then requests for RLF software should be made to:

```
RLF
Paramax STARS Center
12010 Sunrise Valley Drive
Reston, VA 22091
```

New releases of RLF can also be obtained in this way. Joining the **rlf@stars.rosslyn.paramax.com** mailing list will keep you apprised of new releases or bug fixes.

6.2 Doing Software Updates

Each full release or bug fix release of RLF is accompanied by a Version Description Document (VDD) and one of either the **RLF Source Code Release Installation Guide** or the **RLF Binary Release Installation Guide**. The VDD identifies the software release and also notes any changes which have occurred since the last release. It will also make suggestions about how much of RLF needs to be re-installed to meet the release. The Installation Guide gives specifics on how to install RLF by building all the source code or by just installing new application executables.

6.3 Reporting Errors

If errors in RLF applications or documentation are encountered, then a Program Problem Report should be filled out and sent by electronic mail to **rlf-bugs@stars.rosslyn.paramax.com**. If electronic mail is not available, the completed problem report should be mailed by standard post to:

```
RLF
Paramax STARS Center
12010 Sunrise Valley Drive
Reston, VA 22091
```

When the completed Program Problem Report is received, it will be acknowledged and the problem will be handled.

Perceived deficiencies in RLF may not necessarily be errors and warrant a Program Problem Report. If the RLF is performing as documented but is still thought to be deficient, then a New Feature Request form should be completed and submitted in the same way in place of a problem report.

6.4 Getting Help

Requests for assistance with RLF are best handled through a Program Problem Report submitted to **rlf-bugs@stars.rosslyn.paramax.com** if RLF is having errors, or by electronic mail to the RLF mailing list, **rlf@stars.rosslyn.paramax.com**, if there is a lack of understanding or some questions. If electronic mail is unavailable, writing to the standard mail address in section 6.3 will provide assistance.

A Summary of the Library_Manager Application

A.1 Overview

The **Library_Manager** tool performs tasks associated with library administration. These include manipulating entire RLF reuse libraries built from LMDL library model specifications, browsing and examining the contents of reuse libraries, and performing a small number of manipulation on the parts of libraries. The **Library_Manager** only supports limited library model editing including the deletion of attributes and inferencers from a model. Any editing of this sort done with the **Library_Manager** should be also be done in the LMDL specification of the library model in order to maintain integrity between the specification and the model, although the using the **Library_Manager** allows the changes to be made without rebuilding the entire model at that time.

The **Library_Manager** currently operates on one directory of library model instances at a time. Selections to load or delete libraries are made from this directory, which can be set in the RLF start-up file `.rlfrc`, with the **RLF_LIBRARIES** environment variable, or on the command line with the **-I** option. Within any loaded library, the **Library_Manager** maintains the idea of a "current category" which provides the context for all operations. An object can be navigated to through the library model hierarchy and can serve as the current "category."

A.2 Layout

The **Library_Manager** is a window application which contains a menu bar, title bar, and text window. The menu bar contains four buttons which have pulldown submenus and one quit button. Initially only the "Library" and "Quit" buttons will be sensitized for user selection. The "Browse", "Edit", and "Asset Management" menu buttons will be sensitized after a library is loaded. The title bar contains the current directory and library information. The directory information will be displayed upon start-up but the library information will not be displayed until a library is loaded by the user. The text window is scrollable and is used to display information about the current category, including lists of its relationships, actions, or attributes.

A.3 Functionality

This section discusses the functionality of the **Library_Manager** application on a menu by menu, button by button basis. Each subsection discusses each of the entries on the **Library_Manager** main menu bar.

A.3.1 "Library" pulldown menu

The "Library" pulldown menu contains five buttons which operate on an entire library. These are "Load", "Delete", "Close", and "Save". Initially only "Load", and "Delete" are sensitized for user selection. "Delete" will only ever be sensitized when there is no library open. "Load" continues to be available since it acts as a re-load, saving and closing the currently opened library, if selected when a library is already open.

Load Selecting this button causes a scrolled list of library names to pop up. Choosing a library name loads that library and makes its root category the current category. The library name is displayed in the title bar of the window. Information regarding the current category is displayed in the text window. An informative warning dialog box is popped-up if there are problems in opening a library. A "Cancel" button under the scrolled list can dismiss it.

Delete Selecting this button, available only when no library is currently open, also pops up a scrolled list of library names to select. Choosing one deletes a library from the current directory. A confirmation dialog box is popped-up. A "Cancel" button under the scrolled list can dismiss it.

Close This button is only sensitized when there is a library open. Its function is to close a library. Changes are not saved.

Save This button is sensitized when there is a library open. It writes the library to disk but does not close it.

A.3.2 "Browse" pulldown menu

The "Browse" pulldown menu on the menu bar becomes sensitive for selection when a library is loaded. There are two entries in it's submenu, "Navigate Hierarchy" and "Examine Hierarchy". Both of these buttons have a pulldown menu associated with them. "Navigate Hierarchy" should be used to go from one place to another in a model. "Examine Hierarchy" displays information pertaining to the category the user is positioned at.

Navigate Hierarchy This submenu contains buttons for choosing different ways to move through the library model category hierarchy. These are: "User Entered Name", "Go to Parent", "Go to Child", "Follow Relationship", "Backtrack Relationship", and "Go to Action Category". The goal of any of these is to change the current category or object.

User Entered Name This button pops up a dialog box so the user can enter a destination name directly, without proceeding through all the intervening categorys, using the other navigation buttons.

Go to Parent This button produces a submenu of all the current category's parent category(s). Choosing one will make it the current category.

Go to Child This button produces a submenu of all the current category's child categories or objects. Choosing one will make it the current category.

Follow Relationship	This button produces a submenu of all the current category's relationships. Choosing one will make the relationship's filler type category the current category.
Backtrack Relationship	This button produces a submenu of all the relationships which have the current category as a filler type. The owner of each relationship is also shown. Choosing one will make the relationship's owner category the current category.
Go to Action Category	This button produces a submenu of all the action categories associated with the actions at the current category. Choosing one will make it the current category.

Examine Hierarchy This submenu contains buttons for displaying the following information: "Display Relationships", "Display Actions", "Display Attributes", and "Display Inferencer". When any of these are selected the appropriate information will be displayed in the text window.

Display Relationships	This button causes all of the relationships associated with the current category to be displayed in the text window.
Display Actions	This button will not be sensitized unless there are action(s) associated with the current category. Selection of this button causes a description of each action to be displayed in the text window.
Display Attributes	This button will not be sensitized unless there are attribute(s) associated with the current category. Selection of this button causes the list of text and integer attributes to be displayed in the text window.
Display Inferencer	This button will not be sensitized unless there are inferencer(s) associated with the current category. Selection of this button causes the name of the associated inferencer to be displayed in the text window.

A.3.3 "Edit" pulldown menu

The "Edit" pulldown menu contains buttons for "Delete Attribute" and "Delete Inferencer".

Delete Attribute This button causes all of the attributes associated with the current category to be displayed in a submenu. Selection of one of these attributes will cause the attribute to be deleted. Upon reentering the submenu it will not be apparent that the deletion has taken place but the change will be reflected if the library is reloaded.

Delete Inferencer This button causes all of the inferencers associated with the current category to be displayed in a submenu. Selection of one of these inferencers will cause the inferencer to be deleted. Upon reentering the submenu it will not be apparent that the deletion has taken place but the change will be reflected if the library is reloaded.

A.3.4 "Asset Management" pulldown menu

The "Asset Management" pulldown menu contains buttons for "Extract Asset", "Export Asset", and "Import Asset".

Extract Asset This button will invoke the built-in `Extract` procedure action which will copy all the file attributes at the current category to the location as given by the `RLF_WORKING_DIR` environment variable. If this variable is not set, this action will use the current working directory. The ability to copy the associated files depends on the file permissions set in the operating system.

Export Asset This button writes information about the asset at the current category into a file in the working directory so the asset can be imported into another reuse library. This function is not fully implemented.

Import Asset This button reads information about an asset in a file in the working directory, constructs an appropriate category for it in the library, and enters the asset in the reuse library. This function is not fully implemented.

A.3.5 "Quit" button

There is a button labeled "Quit" of the `Library_Manager`'s main menu bar which exits the application. Quitting with a library loaded will first save and close the library. Closing the library, or saving then closing the library, will greatly reduce the amount of time it takes the application to quit. When the "Quit" button is selected, A confirmation dialog box will appear to make sure the user wishes to quit.

B .rlfrc Start-Up File Syntax Summary

B.1 Notation

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

lower_case_word
: nonterminal (e.g. library_model_spec).

*italicized_part*_lower_case_word
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *category*_name).

bold_face_word
: language token (e.g. **category**).

`{item}`
: braces enclose item which may be repeated zero or more times.

`[item]`
: brackets enclose optional item.

`item1 | item2`
: alternation; either item1 or item2

B.2 .rlfrc File Syntax

```
startup_file ::=
      {setting}
```

setting ::=
default_directory |
default_library |
start_category |
view_type |
view_depth |
topology_flag |
cardinality_flag |
layout_offset |
bitmap |
tau_setting |
debug_flag |
working_directory |
history_list_length |
default_editor |
default_pager |
translator_setting

default_directory ::=
library directory : pathname

default_library ::=
library : name

start_category ::=
initial category : name

view_type ::=
view type : agg_or_spec

agg_or_spec ::=
relationship | specialization

view_depth ::=
view depth : [agg_or_spec :] depth_setting

depth_setting ::=
all | integer

topology_flag ::=
topology : flag_setting

flag_setting ::=
yes | no | true | false | on | off

cardinality_flag ::=
cardinality : flag_setting

layout_offset ::=
layout offset : [x_or_y :] integer

x_or_y ::=
x | **y**

bitmap ::=
node bitmap : category_or_object
[: has_attribute {has_attribute}] : pathname

category_or_object ::=
category | **object**

has_attribute ::=
inferencer | **actions** | **attributes**

tau_setting ::=
advice : tau_setting_type

tau_setting_type ::=
explanations : explanation_type |
automatic move : flag_setting

explanation_type ::=
none | **all** | explanation_kind {explanation_kind}

explanation_kind ::=
reasoning | **questions** | **moving**

debug_flag ::=
debug : flag_setting

working_directory ::=
working directory : pathname

history_list_length ::=
history length : integer

default_editor ::=
editor : string

default_pager ::=
pager : string

translator_setting ::=
translator : translator_type

translator_type ::=
  lmdl : lmdl_setting |
  rbdl : rbdl_setting

lmdl_setting ::=
  quiet_translation | translate_only | default_input_spec

rbdl_setting ::=
  quiet_translation | default_input_spec

quiet_translation ::=
  quiet : flag_setting

translate_only ::=
  only : model_or_state

model_or_state ::=
  model | **state**

default_input_spec ::=
  default specification : pathname

integer ::= digit {digit}

name ::= identifier | " character {character} "

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

pathname ::= printable_non_whitespace {printable_non_whitespace}

B.3 Example .rlfrc File

```
--|
--| Sample startup file for the Reuse Library Framework version 4.0
--|

--|
--| Library directory or name specifications
--|
--library directory : /path/Libraries
--library : "Sort and Search Algorithms"
```

```
--|
--| Parameters for the RLF Graphical Browser
--|
topology : off
cardinality : off
layout offset : x : 20
layout offset : y : 5
history length : 50
view type : specialization
view depth : relationship : 2

--|
--| AdaTau inferencing settings
--|
advice : explanations : all
advice : automatic move : false

--|
--| Bitmaps for nodes
--|
--node bitmap : category : /path/box_m.xbm
--node bitmap : category : inferencer : /path/box_I_m.xbm
--node bitmap : category : actions : /path/box_A_m.xbm
--node bitmap : category : inferencer actions : /path/box_AI_m.xbm
--node bitmap : object : /path/cube_m.xbm
--node bitmap : object : inferencer : /path/cube_I_m.xbm
--node bitmap : object : actions : /path/cube_A_m.xbm
--node bitmap : object : inferencer actions : /path/cube_AI_m.xbm

--|
--| Specification translator settings
--|
translator: Lmdl: quiet: no
translator: Rbdl: quiet: no
```

C PCTE and RLF

In most respects, the PCTE version of this delivery of RLF will operate in the same manner as the UNIX version. This appendix, however, will present the differences in the PCTE and UNIX versions of RLF and present some conventions which can be used to produce library models which will be portable between versions. It will also list some requirements of the PCTE version which are not UNIX requirements. This appendix assumes knowledge of PCTE, the Emeraude PCTE product, and the *esh* shell.

C.1 File Naming Restrictions

The Emeraude implementation of PCTE places restrictions on the length of object names and makes assumptions about the use of '.' in object names. The names of files containing assets which are available in an RLF reuse library are restricted to 32 characters in length when using PCTE. These are the files that reside beneath the `Text/` subdirectory of any directory where RLF libraries have been constructed. Additionally, the names of files containing reusable assets in the library should not contain the '.' character, since this indicates a special meaning to the Emeraude implementation of PCTE. The convention established by this version of RLF for PCTE is to replace any '.' characters in file names with the underscore character, '_'. An exception to this convention is the `.rlfrc` start-up file, which the PCTE version of RLF will look for as an entity named `rlfrc.e`.

To increase the similarity in the way libraries are represented in the UNIX and PCTE versions, and to ease transition between versions, the preferred link type of every object in or beneath the directory object where the library was built must be set to ".e". This includes files representing a library's assets and any action scripts which might appear below the `Text/` directory. The preferred link type of the directory object indicated by the environment variable, `RLF_LIBRARIES`, also needs to be ".e" so that its subdirectories can be traversed easily.

Library representations built with the PCTE version of RLF also require a directory object named `rlf_tools` to be a first-level subdirectory of the directory object where the library is built. This directory object must also contain two tools named `ascii_file.tool` and `displ_attr.tool`. These tools are required for RLF's default actions to operate correctly.

For excellent examples of library model construction for the PCTE version of RLF, examine the `.esh` versions of the build scripts for the example libraries delivered with RLF. These scripts are found in each subdirectory of the `models/` directory of an RLF installation. These scripts can be modified and reused to help automate the procedures required to build an RLF reuse library with the PCTE version.

C.2 Action Modeling with PCTE

The modeling of "`System String`" type actions in PCTE has stricter requirements than for UNIX. (See section 4.3.2 for details on UNIX action modeling.) In the UNIX version, commands may be placed directly in the string attribute of the action category, and then this command will be executed in its own UNIX shell when an action which references it is invoked. PCTE, however, must invoke an *esh* process to perform actions. Because of this, all "`System String`" type actions in the PCTE version of RLF must be in an *esh* script which can be executed in a PCTE process. This is very similar to encapsulating an action in a *csh* script which is executed by UNIX. Additional parameters may be supplied just like for any other *esh* script, but no pipes, output redirection, backgrounding of the task, or multiple commands (colon separated list of commands) are allowed in the string attribute which represents the action at the action category. If any of these capabilities are needed for the action to meet its goals, then these things should be done within the *esh* script which performs the action. Also, within the script which performs the action, if it is necessary to execute UNIX-only, non-encapsulated programs (e.g. **xloadimage**), then the script will have to use the *esh* command **epath** to convert a PCTE pathname to a UNIX file name.

The PCTE version of RLF assumes these *esh* action scripts can be found in the **Text/** subdirectory of the directory object where the libraries were constructed. (This is the directory usually specified using the **RLF_LIBRARIES** environment variable.) The modeler must specify any additional paths in the LMDL specification. Typically, the scripts are deposited in a model-specific directory object in the **Text/** directory object. For example, if a library describing animals has an action which invoked an *xterm* and ran *less* in it to view an asset, the final location of the *esh* action script might be

```
$RLF_LIBRARIES/Text/animals/xterm_less.tool
```

Scripts must be of type **sctx** and should use the link extension **.tool** in the PCTE object base. The script writer and installer should verify that the execute permissions are set for any action scripts. The command "**obj_set_mode a+x <pathname>**" executed in *esh* would do this for the script indicated by **<pathname>**. If the script is created as a UNIX file and has the correct permissions in UNIX, the PCTE copy will have the correct permissions.

For a larger example, suppose an RLF library model is being developed which will be a repository of bitmaps. A necessary action for this library is one which allows the user to view a bitmap which is a candidate for reuse. A "**View Bitmap**" action category which might appear like this in the action sub-model in the UNIX version:

```
category "View Bitmap" (Action) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xloadimage ## &";
   end attributes;
end category;
```

In the PCTE version, this action should be modeled like this:

```
category "View Bitmap" (Action) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
```

```
        string is "bitmaps/xloadimage.tool ##";
      end attributes;
    end category;
```

In this example, **xloadimage.tool** is an *esh* script which PCTE will invoke in a separate process. The contents of **xloadimage.tool** might look like this:

```
xloadimage "'epath $1'" &
```

This example illustrates a few of the differences between the UNIX and PCTE versions of RLF. It shows the necessary encapsulation of an action in an *esh* script for PCTE which appears as the action category's string attribute with no pipes, file re-direction, multiple commands, or backgrounding. It also shows that once inside the script, these things can be done as usual, and that UNIX-only tools need to use **epath** to resolve the actual UNIX pathname of a PCTE object with contents. Using the action script location convention, this script would be located in

```
    $RLF_LIBRARIES/Text/bitmaps/xloadimage.tool
```

where **RLF_LIBRARIES** indicates the directory object where the library was constructed and **bitmaps/** is a model-specific subdirectory of **Text/** where files related to the bitmaps library will reside.

This example also shows how UNIX/PCTE portable library models can be developed if useful. The PCTE version of the **"View Bitmap"** action above would work equally well for UNIX if a *csh* script named **xloadimage.tool** was written with the following contents:

```
#! /bin/csh -f
xloadimage $* &
```

If appropriate versions of **xloadimage.tool** were installed in the library directories according to which version of RLF was being used, and file names for assets and related files were carefully chosen (according to the guidelines above) for the UNIX version, then the production of UNIX/PCTE portable library model specifications is not difficult.